Unedited Version. In: Sustainable ICTs and Management Systems for Green Computing. IGI Global. 2012

# Architecture and Mechanisms of Energy Auto-Tuning

**Sebastian Götz, Claas Wilke, Sebastian Cech and Uwe Aßmann**
*Technische Universität Dresden, Software Technology Group, Germany*
*sebastian.goetz@acm.org, {claas.wilke, sebastian.cech, uwe.assmann}@tu-dresden.de*

## ABSTRACT

Energy efficiency of IT infrastructures has been a well discussed research topic for several decades. The resulting approaches include hardware optimizations, resource management in operating systems, network protocols and many more. The approach we present in this chapter is a self-optimization technique for IT infrastructures, which takes hard- and software components as well as users of software applications into account. It is able to ensure minimal energy consumption for a user request along with a set of non-functional requirements (e.g., the refresh rate of a data extraction tool). To optimize the ratio between utility of end users and the cost in terms of energy consumption, the system needs inherent variability leading to differentiated energy profiles and mechanisms to reconfigure the system at runtime. We present our approach called energy auto-tuning (EAT) comprised of these mechanisms and an architecture which automatically tunes the energy efficiency of IT systems.

## INTRODUCTION

Today, developing energy efficient software has become one of the major challenges for software engineering. According to several studies, the carbon dioxide emissions caused by information and communication technology (ICT) were estimated as two percent of world-wide CO2 emissions in 2007 and further increases are expected (Gartner, Inc., 2007) (The Climate Group, 2008). For ICT hardware, like CPUs, hard drives and network devices, several energy optimization techniques have been explored (Hewlett-Packard; Intel; Microsoft; Phoenix Technologies; Toshiba, 2010). However, approaches taking software into account have been investigated in a less intensive way. Representative approaches are (Fei, Zhong, & Jha, 2008), (Kansal & Zhao, 2008) and (Seo, Malek, & Medvidovic, 2008).

In this chapter we propose energy auto-tuning (EAT) as a solution for energy-optimizing software systems. EAT bases on component-based software development (CBSD) and takes both software and hardware components into account. Software components can provide different qualities (e.g., different refresh rates or resolutions) and can be deployed on different hardware devices (e.g., different servers). At runtime, the EAT system decides where to deploy software components once a user requests a provided service. Furthermore, the EAT system is capable to select the best of multiple existing implementations (variants of components) providing different quality of services and consuming different amounts of energy. The optimal variant is selected and deployed at the optimal server w.r.t. its energy consumption.

The basic principle of auto-tuning (AT) can be described as a control loop. An AT system, running on a hardware infrastructure, monitors and controls itself. For each user request, the system reflects if it is able to provide the highest possible user utility for the least possible cost in terms of efforts like energy consumption in its current configuration. A system configuration in our terms is a set of software component instances deployed on component-containers, which run on servers (or, more general, computing entities). To improve efficiency, the AT system does not only need to know its current configuration, but needs to be able to compute other (probably more optimal) configurations and to reconfigure itself at runtime. Notably, reconfiguration is an activity of the system, which implies costs by itself. Hence, the effort imposed by reconfiguration needs to be taken into account additionally. The basic steps of an AT system are depicted in Figure 1.

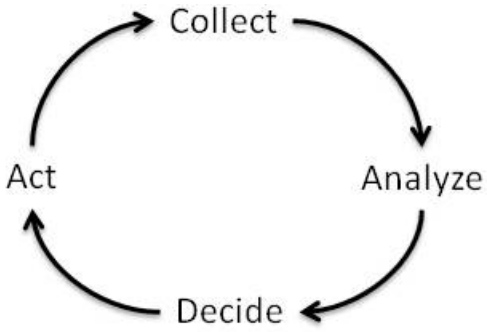


*Figure 1. Control Loop of Auto-Tuning Systems. Adapted from (Dobson, et al., 2006).*

They are: *collect* information about the system in its current configuration, *analyze*, if the system is able to optimally serve the user request, *decide* for a reconfiguration if another, more efficient configuration exists, and *act*, i.e. perform the actual reconfiguration. The general objective function of the overall system is described by Formula 1:

$$efficiency = \frac{\sum_{i=1}^{n} (utility_i \cdot w_i^u)}{\sum_{j=1}^{m} (effort_j \cdot w_j^e)}$$

*Formula 1. General Objective Function for Efficiency.*

Efficiency is defined as the ratio of the sum of weighted utilities and the sum of weighted efforts to achieve these utilities. The number of utilities is denoted by *n*, whereas the number of efforts is denoted by *m*. The variables $w_i^u$ and $w_j^e$ (i.e., weights) are used to prioritize certain utilities and efforts respectively. Energy efficiency is a special case of this formula, where the only effort of interest is the energy consumption (cf. Formula 2).

$$efficiency_{energy}(F,W,C,C') = \frac{\sum_{i=1}^{n}(utility_i(F,C') \cdot w_i^u)}{energy(F,C') \cdot w^e + energy(RC(C,C'),C)} \quad \left| w_1^u ... w_n^u, w^e \in W \right.$$

*Formula 2. Objective Function for Energy Efficiency.*

This formula reflecting our approach considers the user demands (i.e., a set of weights $w_i^u$) provided by the user requesting a system feature *F*, the current system configuration *C* and the target (i.e., possibly new) configuration *C'*. *RC* is a special system feature, which reconfigures the system. It takes the current and target configuration as input. Both utility and energy are functions deriving the achievable utility and energy respectably for a feature on a system configuration. Both take the requested feature and the current system configuration as input. Systems following this objective function are energy auto-tuning systems.

Energy efficiency in our approach is achieved by constantly switching to the system configuration, which has the highest value w.r.t. Formula 2. That is for a system configuration *C*, a specified user request *F* and a set of user demands *W*, another system configuration *C'* is chosen, if the ratio of utility and energy results in the highest value in comparison to all other system configurations. In other words, the achievable utility for the user request in the new system configuration is compared with the energy consumption implied by it as well as by switching from the current to the new configuration. The user demands can be expressed as weights, because they are a set of non-functional requirements, which have to be fulfilled for a user request (functional requirement). To effectively use the formula above, the system needs to derive or know all possible system configurations to compute their efficiency w.r.t. a fixed feature, user demand and current configuration.

To save energy our approach utilizes two techniques: selection and consolidation. Selection denotes the deployment of those software implementations, which use fewer resources compared to other implementations. Because we consider user utility in addition, our approach will not necessarily decide for the least energy consuming implementation, but negotiate the tradeoff between utility and energy consumption as expressed by the formula above. The second technique to save energy is consolidation of workload. As has been shown previously, for example in (Tsirogiannis, Harizopoulos, & Shah, 2010), servers are most energy efficient when they are highly utilized. Hence, our approach derives mappings of software to hardware in such a way that only few resources are used, but these resources are highly utilized. In other words, the work to be performed by the system is consolidated on as few resources as possible. Notably, user utility again poses a tradeoff, which is negotiated by our approach. Using fewer resources typically leads to reductions of non-functional properties (NFPs) (e.g., response time).

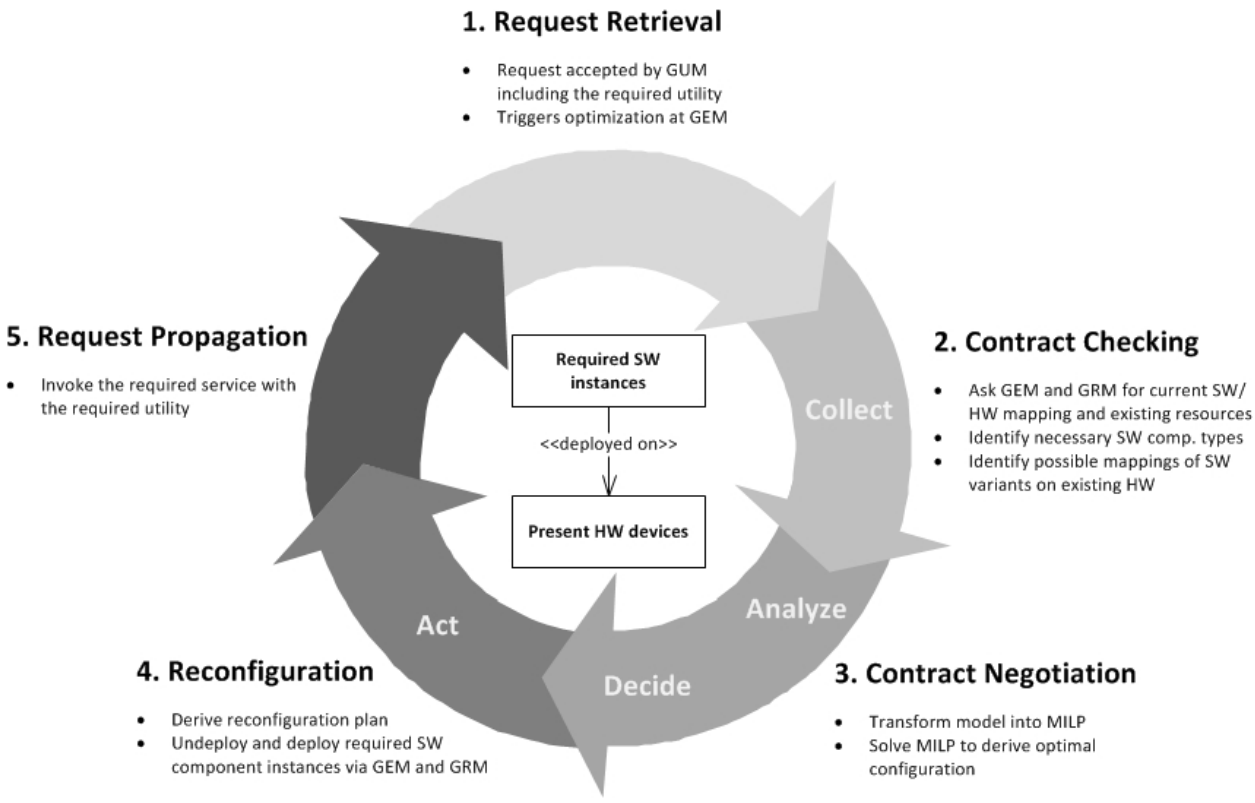


*Figure 2. The specialized Control Loop for Energy Auto-Tuning (EAT).*

To realize EAT we developed the three-layer energy auto-tuning runtime environment (THEATRE) originally presented in (Götz, Wilke, Schmidt, Cech, & Assmann, Towards energy auto tuning, 2010). The runtime process behind THEATRE is depicted in Figure 2. The THEATRE manages a landscape of present hardware components (e.g., servers) on which a set of software components can be deployed and instantiated to provide services that can be requested and invoked by users of the system. The THEATRE consists of three central managers—each representing one of its three layers—maintaining the system and a model that represents THEATRE's knowledge of the system at runtime. The THEATRE receives and executes user requests (user layer, global user manager (GUM)), reconfigures the software (software layer, global energy manger (GEM)) and its deployment on the hardware (resource layer, global resource manager (GRM)) w.r.t. energy efficiency. The five steps of the THEATRE process are shortly outlined below and described in more detail in the remainder of this chapter.

1. *Request Retrieval:* If a user requests a service, the service request is retrieved by the first central manager of the THEATRE, the GUM. Besides the service the user intents to invoke, the user can specify NFPs he requires for the service (e.g., minimum refresh rates).
2. *Contract Checking:* Subsequently, the GUM delegates the request to the GEM to check if the system has to be reconfigured to efficiently serve the user request (e.g., if new software component instances have to be deployed or parts of the system have to be reconfigured to provide the requested NFPs). During contract checking, the GEM searches for the required software components (which can have multiple implementation variants providing different NFPs) and requests the GRM for the currently present hardware devices and their available

resources (e.g., free disk space, memory or CPU cycles). The GEM computes all possible mappings of existing software variants onto the existing hardware landscape. How we model the available software and hardware variants and their dependencies using our cool component model (CCM) and our energy contract language (ECL) is outlined in the next section.

3. *Contract Negotiation:* The requirements of the user request together with the possible software-to-hardware mappings are transformed into a mixed integer linear program (MILP) that is passed to a MILP solver to compute the optimal system configuration w.r.t. the user's demands and the optimal energy consumption rate.
4. *Reconfiguration:* The result from the contract negotiation phase is used to compute a reconfiguration plan denoting which software component variants have to be undeployed, moved, and deployed to provide the requested service and utility. Afterwards, the reconfiguration plan is executed by the GEM.
5. *Request propagation:* The original user request is forwarded to the now energy-optimal deployed software landscape providing the required service and utility.

The attentive reader might wonder how the THEATRE and its managers know about the available software component implementations, hardware devices and how the knowledge about their energy consumption is monitored and maintained to allow the necessary energy consumption prediction for each possible configuration. This process called energy assessment runs in parallel to the EAT control loop and allows to always register new software implementations and to plug or unplug hardware devices.

Besides the presentation of our modeling techniques and the mechanisms of EAT, the remainder of this chapter discusses related and future work as well as challenges regarding our EAT approach.

## ARCHITECTURE OF ENERGY AUTO-TUNING SYSTEMS

In this section we will elaborate on the architecture of THEATRE by discussing the CCM and the ECL. A general overview of the concepts of both the CCM and the ECL is given in Figure 3. The CCM is used to describe the structure of both software (SW) and hardware (HW) components. Furthermore it allows the definition of NFPs provided by different SW and HW component types. In addition, the CCM allows for the definition of behavior for SW and HW components w.r.t. their utilization and energy consumption. Finally, the CCM is used to model individual implementations of SW and HW component types (i.e., variants). The ECL is used to describe provided and required NFPs of individual variants as well as their dependencies to other SW and HW component types. A detailed discussion of the CCM and ECL concepts is given in the following subsections.

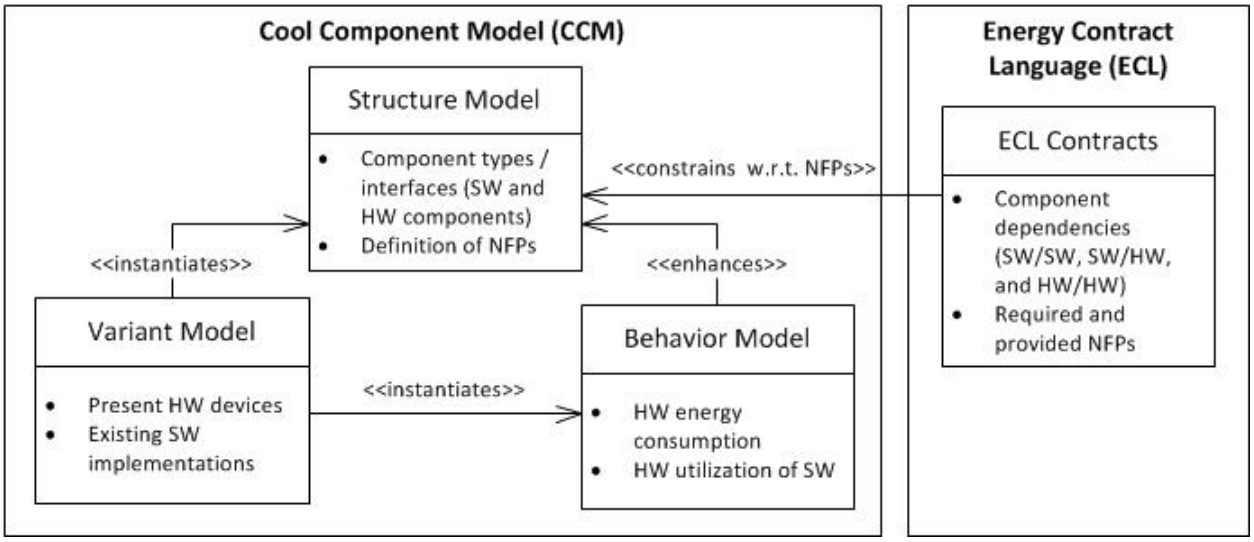


*Figure 3. The Concepts of CCM, ECL and their Interconnections.*

## The Cool Component Model

Component-based software development (CBSD) provides a structured way to construct modular software systems. Szyperski et al. define a software component as "… a unit of composition with contractually specified interfaces and explicit context dependencies only" (Szyperski, Gruntz, & Murer, 2002). In consequence, components explicitly define what they provide to their environment (i.e., interfaces) and in turn what they require (i.e., context dependencies). Component models extend this basic definition by further constraints on what constituents a component needs to comprise and how components can be composed. Our CCM is one such extension, which focuses on variability in terms of multiple implementations (i.e., variants) of component types and the modeling of NFPs, especially energy consumption. Moreover, the CCM does not restrict components to be software artifacts, but allows for the modeling of hardware resources as components, too. Another key characteristic of the CCM is its explicit meta-level. That is the ability to define component types describing components. For software components this meta-level is well-known as it allows for the declaration of multiple implementations per component. For hardware resources the meta-level allows to define types of resources that may exist in an IT infrastructure, like servers, hard disks, but also mobile phones or service robots. The CCM provides three kinds of models to capture an EAT system: structure, variant and behavior models. Structure models describe the aforementioned meta-level. Instances of this meta-level are described by variant models, which allow for the declaration of component implementations and concrete hardware landscapes. Finally, behavior models are responsible to capture the behavior of components w.r.t. energy consumption. Their behavior is expressed in terms of templates, which are based on energy state charts. In the following subsections we will first outline the structure and variant models, followed by a discussion about the modeling of NFPs and the concept of energy behavior templates.

### Structure and Variant Models

The CCM distinguishes between component types and variants in terms of structure and variant models respectively. To provide an overview of these models, we introduce a simple stock tracking application as

a running example in this chapter. This example scenario will be used in the next section to explain how certain EAT mechanisms work.

The stock tracking application's task is to extract and analyze stock quotations from several stock exchanges, to analyze the quotations and to present the results. Figure 4 shows the structure model capturing the software structure of this application consisting of software component types, port types and connector types between port types of different software component types. In the following, elements of the meta-level are formatted in typewriter, whereas variants and NFPs are formatted in italic font. The `Extractor` type collects a history of stock quotations from several companies. It is used by the `Analyzer` type, which may provide several algorithms to analyze arbitrary data. In turn the `Analyzer` type is used by the `Presenter` type to generate a graphical representation of the analyzed data. For each software component type several NFPs are specified that are required for EAT at runtime. The implementations of one software component type have the same NFPs, which differ in their values. For example, each `Extractor` implementation has a *refresh rate*, but its actual value differs for the different variants. Details regarding these NFPs are explained in the next subsection.

For each software component type, several implementations (i.e., variants) exist, which are shown in Figure 4, too. Concrete variants refer to their type and provide values for their functional and non-functional properties. In our scenario there are two variants of the `Presenter` type, an *Image* presenter and an *Interactive* presenter. The first provides a simple image covering the analyzed data, whereas the latter provides additional zooming functionality to the user. For the `Analyzer` type there is a *Standard* variant, which is able to analyze historical data and a *Forecasting* variant, which provides additional forecasting abilities for stock quotations. Finally, a *SingleStock* and a *MultiStock* extractor exist that differ in the number of stock exchanges they can utilize to retrieve stock quotation data. The *SingleStock* extractor is able to collect data from one stock exchange center only (e.g., Frankfurt). In contrast, the *MultiStock* extractor takes several stock exchange centers into account (e.g., Frankfurt, Tokyo and New York).

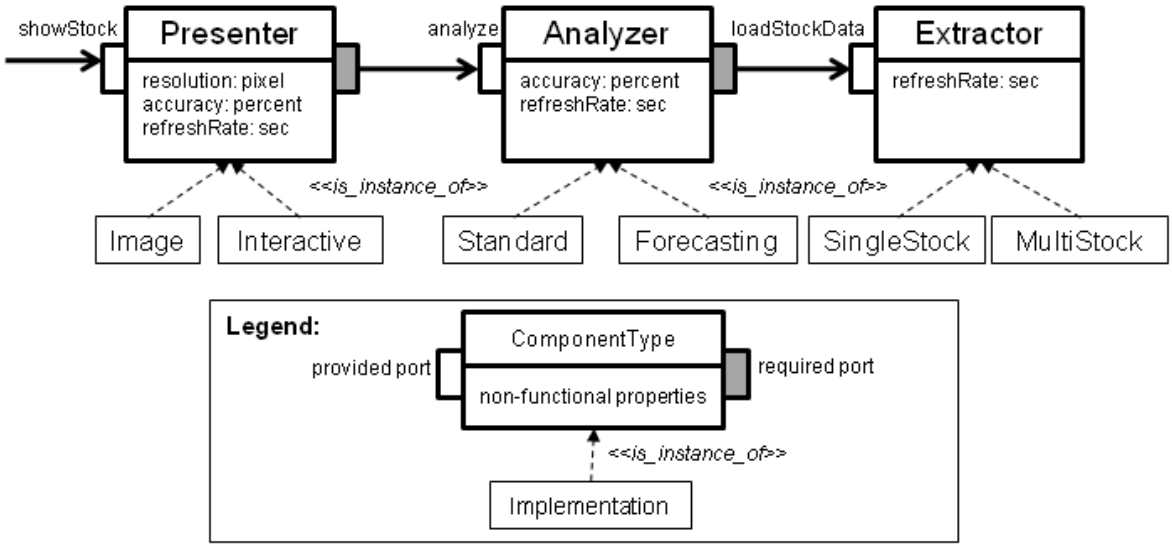


*Figure 4. Structure and Variants of the Stock Tracking Application Scenario.*

For energy-aware optimization, the resources of an IT landscape need to be captured, too. This is because they are utilized by the execution of software components. In other words, software consumes energy by utilizing specific hardware resources during its execution. The IT infrastructure is modeled similar to software component types. There is a structure model describing types of resources that may be part of an infrastructure and a variant model describing concrete resources (variants of a resource type).

The structure and variant model for our example are depicted in the upper and lower part of Figure 5 respectively. The structure model defines that an IT infrastructure consists of one or more servers. Each `Server` may have one or more `CPU`s, network cards (`Net`), `RAM` chips and hard disks (`HDD`). For `Net` and `Server` two port types are specified and connected via port connector types. This indicates that each `Server` is able to connect to other `Server`s to exchange data. Similar to the modeling of software component types several hardware-specific NFPs are defined in the structure model (e.g., the `Net` device provides a *bandwidth*). The concrete infrastructure consists of two server resources. *Server 1* and *2* are variants of the type `Server` defined in the structure model (for clarity the *is_instance_of* relationship is not shown in Figure 5). According to the structure model, `Server`s have to have at least one concrete `CPU`, `Net`, `RAM` and `HDD` resource each. However, they can differ in the concrete values of their NFPs.

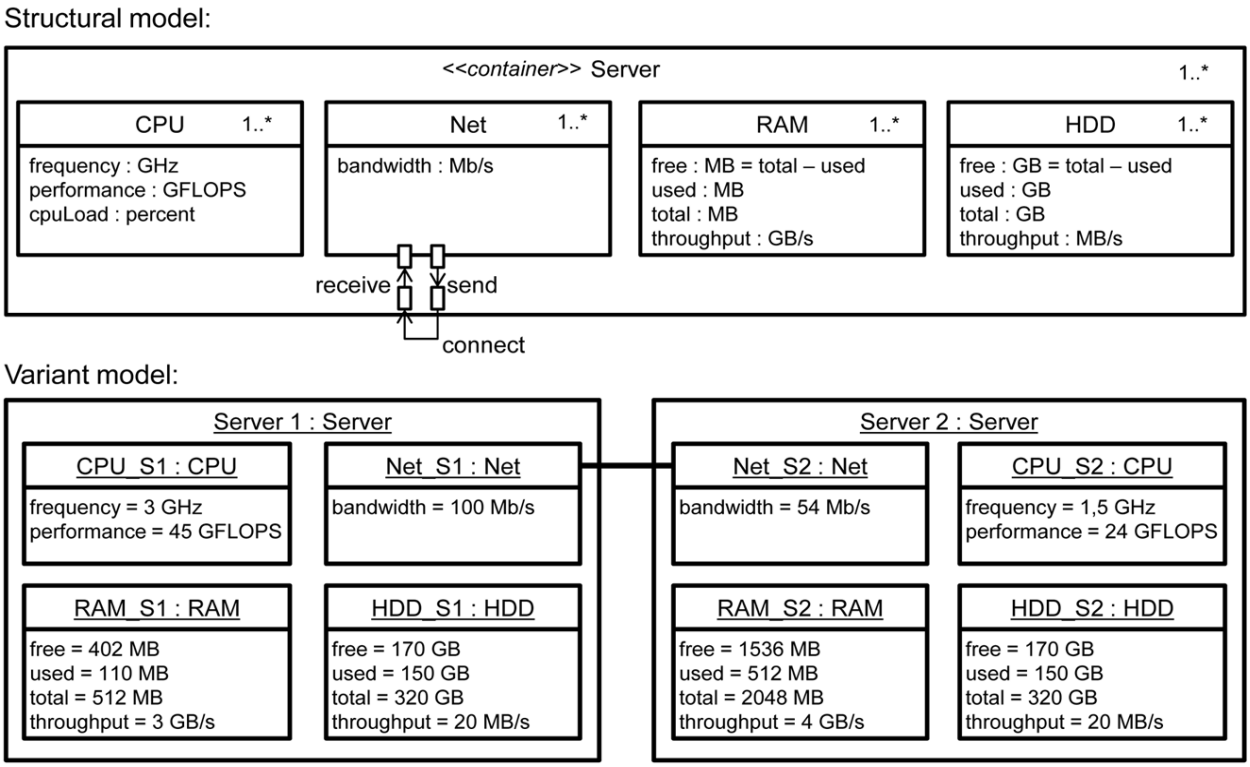


*Figure 5. Structure Model of an IT Infrastructure.*

Notably, any kind of IT infrastructure can be captured by a structure model. A mobile phone would be modeled quite similar to servers. The `HDD` might be replaced by a memory stick and further resource types, like a Bluetooth interface and a GPS device, might be added. A central concept in our approach is the mapping of software component implementations to resources. Therefore, entities of structure models can be marked as containers. In Figure 5 the server is marked as such. In a structure model, which

contains mobile phones in addition to servers, both servers and mobile phones would be marked as containers. As structure models can be defined by the developers of EAT systems, our approach, hence, offers a high degree of flexibility and extensibility w.r.t. the kind of system entities.

## Non-functional Properties

The modeling of NFPs is a key requirement to optimize energy efficiency of complex HW/SW systems. We refer the interested reader to (Chung, Nixon, Yu, & Mylopoulos, 2000) and (Chung & do Prado Leite, On non-functional requirements in software engineering, 2009) for a discussion about functional and non-functional properties in software engineering. Energy consumption is a specific NFP of hardware components, getting aggregated for software components as an NFP, too. Considering our stock tracking application introduced above, further examples of NFPs are the *refresh rate* provided by the `Extractor`, the *accuracy* of the `Analyzer` and the *resolution* provided by the `Presenter` component. In general, each NFP refers or belongs to a component. The *refresh rate* is a property of an `Extractor` component and the *resolution* a property of a `Presenter` component. Hence, in the CCM, NFPs are defined for component types in structure models. They usually have a unit and an increasing or decreasing ordering relation (not shown in our example for clarity). For example, the *refresh rate* for stock quotations is measured in seconds and has a decreasing order (i.e., the shorter the *refresh rate*, the better the utility). The *accuracy* of the `Analyzer` is measured in percent and has an ascending order (i.e., the higher the *accuracy*, the better the utility). As presented in Figure 4 and Figure 5, such properties can be defined for soft- and hardware component types. Typical properties of hardware resources are the total *size* of a hard disk drive, the maximum *bandwidth* of a network device and the *performance* of a `CPU` (e.g., measured in floating point operations per second, FLOPS).

Notably, NFPs are defined on the meta-level (i.e., for hard- or software component types). Their actual values can be determined at runtime for a specific instance (resource or implementation) only. For example, the maximum *throughput* of a network device is specific for each concrete device and the size of an `HDD` is specific for each concrete `HDD`. We distinguish three types of NFPs in the CCM: static instance, monitored and calculated properties. If a value of an NFP is immutable for a given hard- or software instance, we call it a static instance property. The *size* of an `HDD` is an example for this kind of property. If a value of an NFP can only be derived at runtime by profiling the whole (or a part of the) system, we call it a monitored property. In our example, the remaining *free* disk space of an `HDD` is an example for this kind of property. Finally, if a property's value can be derived from other properties, we call it a calculated property. For example, the *used* disk space of an `HDD` can be derived by subtracting the *free* disk space from the (*total*) size of the `HDD`. All three kinds of NFPs can be realized using the same technique to retrieve their value: calling the runtime environment, which is responsible to return the static instance and monitored values as well as to perform required calculations.

## Energy Behavior Templates (Parametric Energy State Charts)

To model the behavior of hardware resources we use the concept of energy state charts (ESCs), as introduced in (Benini, Hodgson, & Siegel, 1998). ESCs base on the existence of power saving or performance modes of hardware resources and extend classical state charts (Harel, 1987) in two ways. Each state and transition is qualified by an energy consumption rate. Transitions are additionally qualified

by a delay. Take the power saving modes of an HDD as an example. A simplified energy state chart for an HDD, as depicted in Figure 6, describes an *idle* mode, a *busy* mode and a *sleep* mode. The HDD will consume more energy if it is *busy*, than when it is *idle* or even *asleep*. Notably, if a read or write request (*rw*) needs to be handled and the HDD is *asleep* it takes time and energy to spin the disk up (i.e., to switch from *sleep* to *busy* mode). A switch from *idle* to *busy* mode will take fewer time and energy.

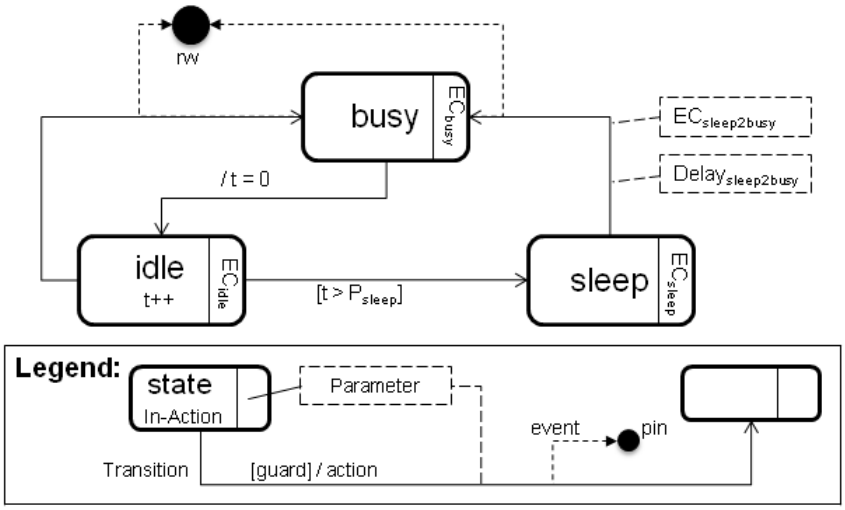


*Figure 6. Energy Behavior Template for an HDD.*

The energy consumption rate and the delays are specific to each concrete resource. But the general behavior defined by an energy state chart can be used for multiple resources. In consequence, the CCM contains energy behavior templates. These templates are energy state charts, which have placeholders for energy consumption rate, delays and custom defined placeholders. In Figure 6 the energy consumption rate (EC) is depicted by the parameters $EC_{busy}$, $EC_{idle}$ and $EC_{sleep}$ respectively. The delay of a transition as well as the corresponding energy consumption rate is depicted in Figure 6 for clarity only for the transition from sleep to busy by the parameters $EC_{sleep2busy}$ and $Delay_{sleep2busy}$.

Declaring an instance of a resource type includes the selection of an energy behavior template and to provide values for the placeholders (cost parameters). These values are not necessarily just numbers, but can be mathematical expressions using the variables defined in the energy state chart as well as the declared cost parameters. A behavior template for an HDD could include a variable counting how often the disk has been spun up or down. This variable could be used in a mathematical expression to calculate the power consumption in a certain state of the state chart to consider the mechanical wear.

The purpose of energy behavior templates is to derive the energy consumption implied by a specified workload. For that purpose, we built a simulation tool, which will be presented in the "Mechanism of Energy Auto-Tuning" section.

## The Energy Contract Language

As mentioned above, component types (software as well as hardware) can define NFPs declaring the qualities they provide at runtime. For each implementation of a component, these values are either bound to concrete values or these values are computed at runtime. However, to configure and auto-tune an

application at runtime the dependencies between component types have to be declared in addition. For example, for a `Presenter` implementation it has to be declared which NFPs of the `Analyzer` component are required to provide certain *accuracy*. To express such dependencies we developed the energy contract language (ECL). The ECL allows for the definition of contracts for components specifying (probably) multiple quality modes providing and requiring different NFPs (i.e., qualities).

We refer the interested reader to (Beugnard A. , Jézéquel, Plouzeau, & Watkins, 1999) for a fundamental discussion on contracts in the context of component-based systems.

```
1  contract Image implements software Presenter {
2
3   mode lowQuality {
4    requires component Analyzer {
5        accuracy min: 0.5
6        refreshRate min: 300
7    }
8
9    requires resource CPU {
10       frequency min: 400
11   }
12
13    provides accuracy min: 0.5
14    provides refreshRate min: 300
15    provides imageWidth min: 800
16    provides imageHeight min: 600
17  }
18
19  mode highQuality {
20    requires component Analyzer {
21       accuracy min: 0.9
22       refreshRate min: 60
23   }
24
25    /* More requirements and provisions here ... */
26  }
27 }
```

*Listing 1. Example ECL Contract for the Image Variant of the Presenter Component.*

Listing 1 shows an example ECL contract for the *Image* variant of the `Presenter` component. The contract consists of two different quality modes (formatted in typewrite in the following), a `low` and a `highQuality` mode (the latter not shown completely). Within each mode, three different kinds of clauses can be specified:

1. Requirements on other software components (e.g., lines 4 – 7 express that the `lowQuality` mode requires an `Analyzer` implementation with an *accuracy* of at least 50% and a *refreshRate* of at least 300 seconds).
2. Requirements on other hardware components (e.g., lines 9 – 11 express that the `lowQuality` mode requires a `CPU` with a *frequency* of at least 400 MHz).
3. Provided NFPs (e.g., lines 15 – 16 express that the `lowQuality` mode provides a resolution of 800 * 600 pixels).

Although not visible in the example contract of Listing 1, all requirements and provisions are typed values as the NFPs are specified using units like seconds or MHz as specified in the structure model. Besides minimum amounts required of specific NFPs, maxima and intervals can be specified as well. Similar contracts as shown for the *Image* variant have to be provided for all software component types' implementations. Thus, similar contracts exist for the *Interactive* `Presenter` variant, as well as all `Analyzer` and `Extractor` variants. Contracts defined for hardware resources are imaginable as well (although not contained in the provided example). However, hardware components can only declare dependencies to other hardware components as they are located "below" the software within the execution platform. Once ECL contracts exist for all components, users can request a service including an expected quality as shown in Listing 2.

```
1 call Presenter.showStock expecting {
2     accuracy min: 0.5
3     refreshRate min: 200
4 }
```

*Listing 2. Example User Request including Demands.*

Depending on the user requirements and the existing component variants, the EAT system will analyze and auto-tune the application w.r.t. maximum utility for minimal energy consumption. After this section discussed the architecture and modeling of EAT systems, the mechanisms behind EAT as introduced by Figure 2 (cf. page 4) are elaborated in the next section.

## MECHANISMS OF ENERGY AUTO-TUNING

In this section we will elaborate on various mechanisms required by an energy auto-tuning system. As a starting point, we will sketch a mechanism to determine and assess all possible system configurations in terms of user demand fulfillment. Then we will discuss an iterative approach combining energy and utility assessment to select the optimal system variant taking reconfiguration efforts into account. After that, we will discuss the process of determining and performing actions required to reconfigure the system. Finally, we will discuss a mechanism to derive the energy consumption imposed by invoking a system feature (i.e., energy assessment).

### Utility Assessment and Contract Checking

In this subsection we focus on the question how to derive configurations, which are able to serve the user's requests and demands (i.e., utility assessment by contract checking).

To determine valid system configurations in terms of software component implementations mapped to hardware resources, we developed a mechanism we call contract checking following the definition of Quality-of-Service-level contracts by (Beugnard, Jézéquel, & Plouzeau, 2010) along with Meyer's design by contract principle (Meyer B. , 1992). The mechanism can be divided into five steps:

(1) quality mode selection,

(2) resolving software dependencies,

(3) collection of resource requirements,

(4) determination of all SW/HW mappings, and

(5) identification of valid configurations, which are those able to serve the user's request and demands (i.e., can be mapped onto available hardware).

The process is initiated by the user, which sends a request as shown in Listing 2 (cf. page 12) to the system. The demands of this request are compared with the provided NFPs of each contract, which has been defined for the component type the request is referring to. In the example introduced in the last section, step (1) will identify the `highQuality` mode of the *Image* implementation (cf. Listing 1 on page 11) as being able to serve the user demands. The `lowQuality` mode does not fit, because a *refreshRate* of at least 200 seconds has been requested by the user, but the `lowQuality` mode only ensures a minimum *refreshRate* of 300 seconds.

The second step recursively resolves the dependencies to software components as declared in the modes, which have been identified before. In the example, the dependency to the `Analyzer` component will be resolved. This is done by handling the dependency like a user request. That is, software component dependencies are interpreted as calls for which valid modes (step (1)) have to be identified. This is possible, because dependencies between software components specify that the dependent component type will call another component type. The result of resolving all dependencies are quality paths. These are sets of quality modes of each implementation required to serve the user's demands. In the example, each quality path consists of three implementation-quality mode pairs: one mode per `Presenter`, `Analyzer` and `Extractor` implementation. The reason is, that one implementation of each component is required to serve the user's request.

The resource requirements can be directly extracted from the respective contracts of each identified quality path. The determination of all SW/HW mappings is realized by building the cross-product of implementations and resources marked as containers. Which resources actually exist is either modeled statically or can be retrieved from the GRM, which is responsible to manage all resources of the infrastructure. Thus, steps (3) and (4) are straight forward. Finally, step (5) investigates each configuration determined in step (4) w.r.t. the possibility to map the implementations to resources.

Multiple optimizations to this general mechanism are possible. For example, we combined the steps (4) and (5) to skip mappings of which we know that they will not support the user's request, which is possible due to the order in which the mappings are checked. Nevertheless, contract checking has exponential complexity, because each dependency between software components duplicates the number of quality paths by the number of existing component implementations and quality modes of the required software component. Thus, the approach is suboptimal for systems with many dependencies between software components and long dependency chains. Nevertheless, contract checking does not need to be repeatedly performed unless the hard- or software landscape changes. Notably, the approach is iterative, meaning that the addition of new resources or software components does not invalidate the results of existing mappings. The removal of a resource or software component only leads to the removal of all mappings, which include the respective entity, but does not affect any other mapping.

In summary, contract checking is a mechanism to assess system configurations w.r.t. user utility, which filters all configurations, which cannot be deployed on a provided infrastructure. In principle, multiple

working configurations can be identified. Contract negotiation, which will be explained in the next subsection, provides means to order these configurations and thus, to identify the best of them.

## Contract Negotiation

Negotiation of contracts is the central mechanism of EAT systems. Its main task is to identify the optimal system configuration for a given user request and a set of user demands. More generally, contract negotiation needs to consider multiple concurrent user requests and demands as well as all predictable future user requests. We call this mechanism contract negotiation, because it negotiates the tradeoff between high user utility (serving user demands) and low energy consumption by utilizing the specified contracts of the EAT system. Contracts implicitly define this tradeoff by specifying provided and required NFPs. They explicitly state resource requirements and thus, implicitly energy consumption.

To identify the optimal system configuration we use the energy assessment mechanism, which includes the assessment of system reconfiguration efforts, and the utility assessment mechanism presented in the previous subsection. Energy assessment leads to a numerical result, namely the amount of energy, which will be consumed by a specified user request. In contrast, utility assessment results in a set of valid mappings of implementations to container resources. To quantify these mappings in terms of utility, the provided NFPs of each mapping are used, which can be extracted from the contracts identified by the respective quality paths.

The set of all possible system configurations can be described as a constraint system, which has three kinds of variables:

(1) Boolean variables indicating whether or not a software component implementation shall be mapped to a resource,

(2) Resource usage variables and

(3) Software-related NFP variables.

The determination of the optimal system configuration is, hence, a constraint solving optimization problem. The translation between resource usage and implied energy consumption is realized by computed factors of the objective function. The objective function is to minimize resource usage weighted by these factors. The minimum utility requested by the user is expressed by constraints. Fortunately, our optimization problem can be formulated using linear constraints only. Hence, we are able to specify our optimization problem as a linear program. The requirement of Boolean variables in addition to floating point variables classifies our problem as mixed integer linear program (MILP). An introduction to linear programming and related constraint solving techniques can be found in (Nemhauser & Wolsey, 1988). For a more general discussion on algorithms including linear and dynamic programming we refer the interested reader to (Dasgupta, Papadimitriou, & Vazirani, 2007). A detailed elaboration on the implementation of contract negotiation, the concrete objective function and all constraint types can be found in (Götz, Wilke, Cech, & Assmann, 2011). A successful application of linear programming in the context of smart energy grids has been shown in (Ranganathan & Nygard, 2010).

## System Reconfiguration

A further mechanism required by EAT systems is the ability to determine and execute all actions required to reconfigure the system from a source configuration to a target configuration. The source configuration is a set of software component implementations as well as their mapping to the currently underlying IT infrastructure. The target configuration is the optimal system configuration determined by contract negotiation. The source and the target configuration have to be compared in order to derive a reconfiguration plan consisting of atomic reconfiguration steps to be executed. Deriving a reconfiguration plan includes the following steps:

(1) Identification of changes at the level of software components,

(2) Identification of changes regarding the SW/HW mapping, and

(3) Derivation and execution of a reconfiguration plan.

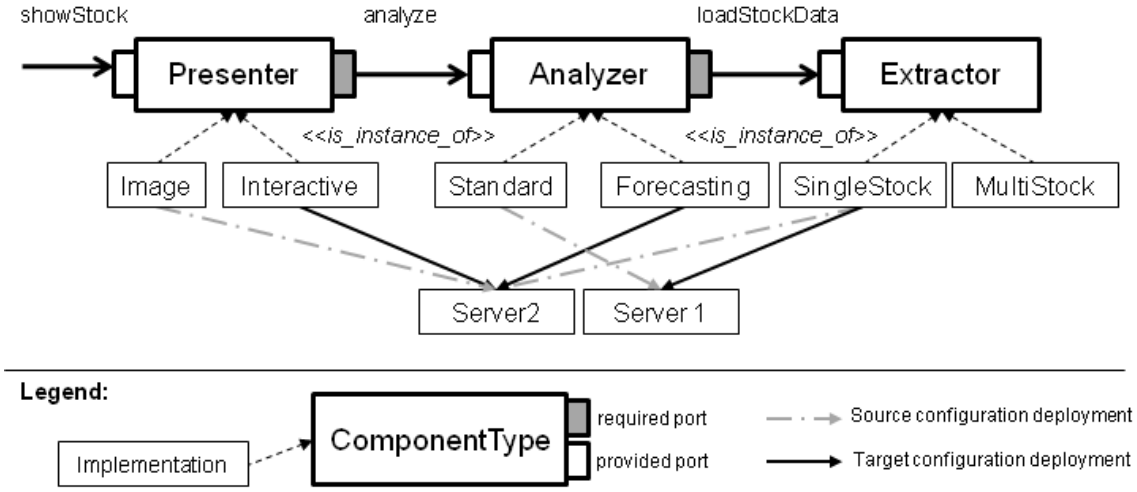


*Figure 7. Example of the Source and Target Configuration of the Stock Tracking Application.*

In the following paragraphs we explain these three steps based on the example introduced in the last section. Figure 7 depicts the available implementations of the three software component types and both servers in the infrastructure. Assume a source configuration consisting of the software component implementations *Image* (`Presenter` type), *Standard* (`Analyzer` type) and *SingleStock* (`Extractor` type). The *Image* and *SingleStock* implementations are deployed on *Server 2*. The *Standard* implementation (`Analyzer` type) is deployed on *Server 1*. In Figure 7 this is indicated by the gray dash-dotted line between implementations and both servers of the IT infrastructure. By using contract negotiation the following software component implementations are detected as target configuration: *Interactive* (`Presenter` type), *Forecasting* (`Analyzer` type) and *SingleStock* (`Extractor` type). The *Interactive* and *Forecasting* implementations have to be deployed on *Server 2* and the *SingleStock* implementation has to be deployed on *Server 1* (indicated by the black bold line between implementations and servers in Figure 7*)*. In terms of the CCM, each configuration can be described as a variant model of software components. System reconfiguration compares both variant models and derives a reconfiguration plan.

In the first step the source and the target variant model are compared w.r.t. their software component types. In cases where the source and the target variant model differ in the selected implementations for the same component type, a reconfiguration step for this component type is necessary. Consider the source and target configuration mentioned above. The first step evaluates that for the `Presenter` type the *Image* variant (provided by the source configuration) has to be replaced by the *Interactive* implementation (provided by the target configuration) and for the `Analyzer` type the *Standard* implementation has to be replaced by the *Forecasting* implementation.

The second step is responsible to discover component types in both configurations for which component migrations are necessary (i.e., which components have to be redeployed on another server). For this purpose each software component implementation in the source and target variant model provides specific deployment information, which is compared to derive the required component migrations. In our example the result is that the implementation behind the `Analyzer` type has to be migrated from *Server 1* to *Server 2* and the implementation of the `Extractor` type has to be migrated from *Server 2* to *Server 1*.

In step (3) the knowledge gathered in step (1) and step (2) is combined in order to extract a reconfiguration plan that describes a set of reconfigurations for each software component type. Such a reconfiguration plan has to be executed by THEATRE to reconfigure the system into an optimal configuration. There are three cases for reconfiguration resulting in three different kinds of reconfiguration steps. These cases are explained based on the reconfiguration plan derived from the source and the target configuration explained above, depicted in Listing 3.

```
1 reconfigurationPlan {
2   reconfiguration for SWComponentType Extractor {
3     migrate SWComponent SingleStock from Server2 to Server1
4   }
5
6   reconfiguration for SWComponentType Analyzer {
7     replace SWComponent Standard on Server1 with Forecasting on Server2
8   }
9
10  reconfiguration for SWComponentType Presenter {
11    replace SWComponent Image on Server2 with Interactive
12  }
13 }
```

*Listing 3. Example of a Reconfiguration Plan.*

The reconfiguration for the `Extractor` type requires that the *SingleStock* implementation has to be migrated from *Server 2* to *Server 1*, because there are only differences w.r.t. the deployment information in the source and target variant model. This information is gathered in step (2). Component migration means that the component state of the *SingleStock* implementation has to be stored on *Server 1*, thereafter *SingleStock* has to be deployed to *Server 2* and finally the component state from *Server 1* has to be restored on *Server 2*. The `Presenter` type requires a local replacement of the *Image* implementation by the *Interactive* implementation. Hence, the component state of *Image* on *Server 2* has to be saved and restored in the *Interactive* implementation after its deployment has been finalized. A local replacement is

necessary in cases where differences between the source and the target variant model only exist at software component level. That is, when the deployment information of both variants of a type is equal, whereas different variants of a software component type are detected in the first step. The last kind of reconfiguration is a remote replacement. It is necessary in cases where a different software component implementation (result of step (1)) and different deployment information (result of step (2)) are recognized. In our example this case occurs for the `Analyzer` type because of the result of step (1) that the *Standard* analyzer has to be replaced by the *Forecasting* analyzer and both implementations have different deployment information, which is determined in step (2).

The employed component containers do not necessarily have to provide the functionality to save and restore the state of software components subject to migration (state migration). The only requirement for containers is their ability to deploy and undeploy software components. State migration and the execution of reconfiguration scripts are realized by the GEM of the THEATRE.

## Energy Assessment

An essential part of energy optimization is to determine how much energy is or will be consumed by the execution of a feature in a certain system configuration or, in short, the process of energy assessment. To determine system configurations, which are better than other configurations in terms of energy efficiency for a given user request and set of user demands, a mechanism to predict the required energy to perform the request on a given system configuration is needed. For that purpose, we developed a simulation approach. Furthermore, we developed an approach to derive the resource usage implied by the execution of a feature, including the reconfiguration as a system feature. To bridge the gap between resource usage and energy consumption simulation, we propose a fine-grained resource usage analysis resulting in workloads for our simulator. In the following we will discuss these steps in detail.

### Simulation of Energy State Charts

To predict future energy consumption we developed a simulator for ESCs. A simulation requires a workload (stimulus) for which the consumed energy (response) shall be predicted. For that reason we introduced two immutable variables to energy state charts per default: total energy consumption (*energy*) and a global timer (*time*). The energy variable is used to accumulate the consumed *energy* over (simulation) time. The *time* represents the current simulation time. ESCs have in- and out-pins. An in-pin denotes an event the ESC is waiting for. Out-pins denote events the ESC will fire. This way multiple ESCs can be connected using corresponding in- and out-pins (e.g., the ESC depicted in Figure 6 on page 10 contains an in-pin for read/write (*rw*) events). The simulation itself is a loop, which stops after the last event of the workload has been processed or at a user-defined point in time. The loop counts the simulation *time* up and checks whether there are enabled transitions starting at the current state. A transition is enabled, if the condition of it evaluates to true, the event it is waiting for is defined in the workload at the current simulation time, and/or neither condition nor (in-)events are defined for that transition. If no transition is enabled, the simulator interprets the energy rate expression defined in the current state and adds the resulting value to the *energy* variable. Else, the transition is fired, the current state is changed to the target state of the transition and the energy consumed by following the transition (delay * energy) is added to the *energy* variable. Notably, the delay of transitions leads to jumps in simulation time. This includes the danger to skip events defined in the workload by accident.

Nevertheless, the workloads on hardware resources are not defined by the user, but are derived using our resource usage analysis mechanism, which will be presented in the next subsection. Thus, events, which occur while a resource is busy by transitioning to another state, are not skipped by accident. Indeed, such events are ignored, because the system is incapable of processing them at the specified point in time.

A drawback of using ESC simulation is their complexity in terms of parallel execution. The main energy consumers in a typical server are CPU, HDD, network devices and, if present, graphical processing units (GPU). These resources are used in parallel and, hence, consume energy in parallel. As indicated before, parallel execution of ESCs can be emulated by connecting their in- and out-pins. That is an ESC fires an event using an out-pin, where another ESC is waiting for that event by an in-pin. The fired events are logged in a global workload. Nevertheless, each ESC has to be simulated in sequence. In the easiest case, an order of interconnected ESCs can be derived. This is done by analyzing the connected in- and out-pins. An ESC which has no out-pins cannot introduce a dependency itself, but can depend itself on another ESC. In the worst case, the ESCs are simulated in time-slices. That is an ESC $\alpha$ depends on an ESC $\beta$, which depends on $\alpha$, too, where a close analysis of the dependency reveals that $\beta$ is waiting for an event of $\alpha$ at time $t_1$ and $\alpha$ is waiting for an event of $\beta$ at time $t_2$ after $t_1$. To simulate both ESCs multiple possibilities exist. First, $\alpha$ can be simulated until $t_2$ followed by a complete simulation of $\beta$ and a simulation of $\alpha$ from $t_2$ to the end. Second, $\beta$ is simulated until $t_1$ followed by $\alpha$, which is simulated till the end and the simulation of $\beta$ from $t_1$ till the end. Both approaches imply similar efforts and enable parallel simulation of ESCs. The determination of the workload required as an input for the simulation will be discussed in the following subsection.

## Resource Usage Analysis of Software Components

The energy consumed by software depends on how the software utilizes which hardware resources. Hence, to optimize the energy efficiency of heterogeneous IT landscapes, our EAT approach requires knowledge about the resource usage of software components during their execution. This knowledge is used to derive a workload for ESCs. In combination with the simulation approach explained above, the energy consumption implied by executing a feature of a software component can be estimated and used as a factor in the objective function of the MILP during contract negotiation. To gather utilization data of certain hardware resources we propose the Java Resource Usage Profiling Infrastructure (JRUPI) (Süttner, 2011) as an approach to extract such data at runtime (i.e., during the execution of software). JRUPI is a flexible and portable framework for profiling and analyzing the resource usage data of arbitrary software component features. Based on collected resource usage data, a mathematical model is derived that describes the resource usage behavior of a certain feature of a software component variant. Assuming Java methods as such a feature, the model is defined by considering the metadata of method parameters as variables. Imagine a list of String objects as a parameter of a method. In this case, variables for the mathematical model might be the mean length of Strings and the length of the list.

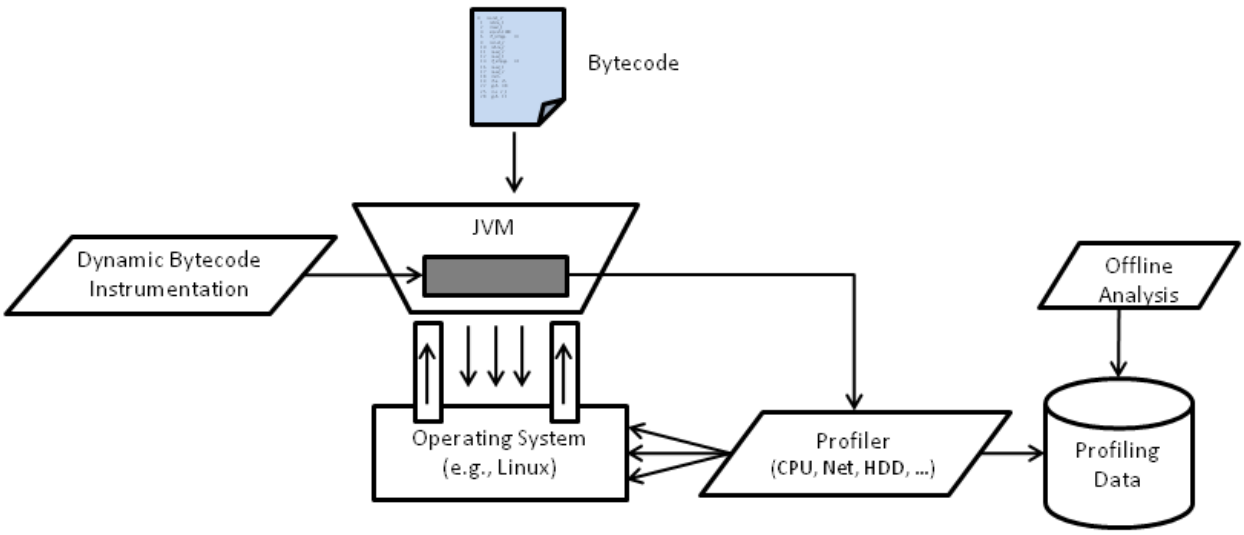


*Figure 8. Resource Usage Profiling with JRUPI. Adapted from (Süttner, 2011).*

JRUPI uses dynamic tracing and profiling techniques (Sun, 2008) to capture resource usage data at runtime. The mathematical model of captured data is derived in a separate offline analysis phase. Figure 8 shows how JRUPI collects resource usage data. At the level of the operating system so called *profilers* are responsible to monitor the resource usage of the Java Virtual Machine (JVM) process. So far, profilers for CPUs, network devices and hard disk drives were implemented by using the dynamic tracing framework SystemTap (Jacob, Larson, Leitao, & da Silva, 2010). Considering the fact that profilers collect resource usage data of the JVM process, all running Java threads are monitored by profilers. However, in our case we are interested in collecting resource usage data w.r.t. a certain software component. This problem can be solved by using dynamic bytecode manipulations techniques. At runtime a software component is represented as bytecode and is executed in a JVM. Like the JVM process is monitored by profilers, the considered software component is monitored, too. The bytecode of methods is instrumented in a manner that at each entry and exit point of a method an event is triggered at the operating system level. This event is captured and evaluated by the system profilers and indicates that subsequent resource usage is correlated to the Java method, which triggered the event. Nevertheless, by using dynamic bytecode instrumentation, profilers are able to selectively monitor events from the considered software component and not from the whole JVM process. From a technical point of view, dynamic bytecode instrumentation is realized using the bytecode manipulation framework BTrace (Oracle, 2010). As mentioned above, an additional offline analysis phase is required to derive a mathematical resource usage model. During offline analysis profiling data is further processed by filtering, aggregating, and calculating statistical parameters. The resource usage model is derived automatically by using the analysis tool Eureqa (Schmidt & Lipson, 2009), which uses symbolic regression to detect formulas in a given set of data. Finally, the resulting (mathematical) formulas are used to derive the workload for ESCs.

To summarize this section, energy auto-tuning systems require several mechanisms to optimize the energy efficiency of IT systems. This includes the ability to estimate the energy consumption consumed by software components for processing a request. Based on this information possible software variants as well as the optimal mapping to hardware resources can be estimated by contract checking and contract

negotiation. In order to realize the optimal configuration at runtime a reconfiguration plan has to be derived and executed.

## RELATED WORK

This section presents related work from the domains of energy auto-tuning, multi-quality component models as well as resource utilization, and energy consumption analysis of software applications. Our general idea of the THEATRE has been elaborated in (Götz, Wilke, Schmidt, Cech, & Assmann, Towards energy auto tuning, 2010). The resource managers of the THEATRE are specified in (Götz, Wilke, Schmidt, Cech, Waltsgott, & Fritzsche, THEATRE Resource Manager Interface Specification v. 1.0, Technical Report TUD-FI10-08, 2010). A detailed introduction into profiling concepts of JRUPI can be found in (Süttner, 2011). A first fully operational proof-of-concept implementation of EAT has been realized by Püschel (Püschel, 2011).

### Auto-Tuning

The term auto-tuning emerged from the field of high performance computing where it typically represents the automatic optimization of an algorithm (e.g., matrix multiplication or Fourier transform) w.r.t. its performance, its input data (e.g., matrix size), and/or its underlying hardware. The major idea behind auto-tuning can be summarized by the auto-tuning control loop as depicted in Figure 1 on page 2. Algorithm implementations with auto-tuning facilities can be distinguished into approaches supporting either optimization at installation time, optimization at runtime, or both. ATLAS (Whaley & Dongarra, 1997) optimizes at installation time by generating optimized code for the underlying hardware considering cache sizes and the amount of available CPU registers. FFTW (Frigo & Johnson, 1998) – a library for Fourier transform – optimizes at runtime by composing several code snippets providing similar functionality with different performance properties depending on the underlying hardware. At runtime, the snippet's performance is analyzed and according to it they are selected and composed to improve execution performance. Whereas ATLAS and FFTW focus on the optimization of specific algorithms w.r.t. their performance, our EAT approach focuses on optimizing complete software applications – probably distributed onto several servers – w.r.t. the tradeoff between energy consumption and user utility (i.e., energy efficiency).

Flinn et al. developed an approach for energy efficient execution of multiple applications on mobile devices. They measured the application's resource usage and derived mathematical models allowing for resource and thus, energy consumption approximation (Flinn & Satyanarayanan, 2004). At runtime, these formulas are used to decide whether to execute an application on the mobile device or on an external server. The approach does not always select the best solution as the decision for the optimal configuration is based on heuristic solvers. Nevertheless, Flinn's approach can be considered as an EAT approach w.r.t. the application's deployment at runtime. However, it is tailored to the local versus remote execution scenario. Our approach is of much broader scope as it considers workload consolidation and migration of implementations between computing entities in general.

Another auto-tuning approach was developed by Lachenmann et al. (Lachenmann, Marrón, Minder, & Rothermel, 2007). They developed the programming language Levels that can be used for the

development of sensor networks. Levels allows defining different levels of quality of service (QoS) for sensors (e.g., data collection and transmission, or only message forwarding from other sensors). The different levels are annotated in the source code and for each such code block the energy consumption rate is annotated as well. At runtime, the system decides which QoS level each sensor should provide w.r.t. its current battery status. This way, Levels can help to improve the lifetime and operability of sensor networks. The Levels approach can be considered as a tailored EAT approach as it focuses on the specific domain of sensor networks.

A further approach for QoS optimization of software applications on mobile devices has been developed by Fei et al. (Fei, Zhong, & Jha, 2008). The development of software applications includes the modeling of different power modes and their provided QoS respectively. The user can prioritize the QoS he is expecting and the operating system schedules the applications w.r.t. the required utility and energy efficiency. Thus, Fei et al. negotiate the tradeoff between energy consumption and user utility similar to our EAT approach. In contrast to our approach, Fei et al. focus on multiple applications on a single mobile device whereas our approach focuses on multiple applications executed on multiple mobile and stationary devices. We expect to gain larger energy savings (e.g., by workload consolidation) as unutilized devices can be powered down during runtime.

Microsoft's research project Cuanta focuses on developing a mechanism to operate clouds of servers in an energy-optimal manner. An investigation how workloads can be consolidated in a server cloud scenario to optimally configure resource utilization w.r.t energy consumption showed that energy-optimal operation is situated between low and high utilization of server resources as low utilization leads to avoidable power consumption during idle phases whereas high utilization leads to energy consumption due to execution delays (Srikantaiah, Kansal, & Zhao, 2008). Furthermore, the work proposes an algorithm to optimally pack workloads on servers w.r.t. performance and energy consumption. After estimating the optimal configuration w.r.t. utilization, workloads are deployed to servers in an energy-optimal manner. Thus, Cuanta can be considered as an energy auto-tuning approach. However, in contrast to our approach software component interaction is not considered. Furthermore, migration and reconfiguration which are essential for auto-tuning are not part of the Cuanta approach.

## Modeling Power Consumption and Multi-Quality Component Models

Besides our CCM, several other research projects have focused on developing component models that integrate NFPs into modeling as well as modeling the energy consumption of (at least hardware) components. A detailed discussion on power modeling in the context of servers is presented in (Rivoire, 2008). In addition, we refer the interested reader to (Aagedal, 2001), which contains fundamental insights into the concepts required in multi-quality component models.

The COMQUAD research project developed a component model including NFP descriptions. Components were separated into their specifications and implementations, to allow modeling of multiple implementation variants for the same component specification (Göbel, Pohl, Röttger, & Zschaler, 2004). Besides, CQML+ contracts (Röttger & Zschaler, 2003) were used to specify NFPs and coarse-grain resource dependencies for NFP-aware scheduling of software components on single-server applications. Both the COMQUAD component model and CQML+ majorly inspired our work on CCM and ECL as well as the design of our resource and energy managers.

The consecutive research projects MADAM and MUSIC developed a component model for self-adaptive applications on mobile devices (Geihs, Khan, Reichle, Solberg, Hallenstein, & Merral, 2006). As this component model supports modeling of multiple implementation variants as well as NFPs, it can be considered as another inspiration of our CCM. In contrast to our approach, MADAM/MUSIC focuses on maximizing user satisfaction (i.e., utility) but does not negotiate the tradeoff between utility and cost.

A third major influence for the development of CCM was the HRC component model emerging from the SPEEDS project (The SPEEDS Consortium, 2009). HRC is currently revised and improved within the CESAR research project (Baumgart, A common meta-model for the interoperation of tools with heterogeneous data models, 2010) (Baumgart, Reinkemeier, Rettberg, Stierand, Thaden, & Weber, 2010) (Armengaud, et al., 2011). It focuses on the component-based development for embedded systems including the capability for both software and hardware modeling and their behavior. HRC uses contracts to specify NFPs. Each contract consists of requirements and provisions and can be aligned to a certain viewpoint (e.g., real-time or safety). Whereas CCM focuses on EAT at runtime, HRC focuses on verification and testing of embedded systems at design time. However, both approaches use contracts to specify NFPs of software and hardware components.

A further, closely related research project is DIVA. The key focus of this project was on the management of dynamic adaptive systems, where especially the problem of exponential growth of potential system configurations has been investigated. DIVA provides a solution by combining methods from aspect-oriented programming/modeling (Kiczales, Lamping, Mendhekar, Maeda, Lopes, & Loingtier, 1997) and MDSD (Morin, Barais, Nain, & Jézéquel, 2009). The DIVA approach allows for automatic adaptation of a system at runtime supporting goal-based optimization of NFPs as well as rule-based reconfiguration of the system (Fleurey & Solberg, 2009). The key difference to our approach is the granularity of the optimization problem. DIVA symbolizes the impact of implementations on NFPs (i.e., the free size of memory is represented by symbols like *LOW*, *MEDIUM* and *HIGH* and the impact of an implementation can only be expressed as being low, medium and so on, too). Our approach supports sub-symbolic information in addition (i.e., the actual value of free size of memory in megabyte). We encapsulate symbolization in our contracts, where quality modes denote the symbols and the expressions on the required NFPs describe the domain of the symbol. Though, reasoning on sub-symbolic information is less efficient, due to the raised complexity, it allows to derive finer-grained configurations (e.g., a configuration could include not just the information which CPU to use, but the (optimal) frequency this CPU should have). Such fine-grained information allows reducing energy consumption in addition to the course-grain decision of which resources to use. Current hardware is usually far from being energy-proportional (Barroso & Hölzle, 2007), which is reflected by a very high baseload electricity and a narrow working area. Imagine, for example, a server consuming 100W being idle and 120W at full load. In consequence, energy savings can mostly be achieved by selecting or turning off the right resources. In such a scenario symbolic reasoning, as in DIVA, is feasible. But especially for the next generation of hardware, which is supposed to be more and more energy-proportional (Borkar & Chien, 2011) there is a need for additional, finer-grained energy optimizations. Therefore, in contrast to DIVA, our approach allows for sub-symbolic as well as symbolic reasoning.

Another modeling approach focusing on NFPs is the Modeling and Analysis of Real-Time Embedded Systems (MARTE) profile (Object Management Group (OMG), 2011) for UML that extends UML with

capabilities to describe NFPs for both, software and hardware components. Typically, NFPs defined in MARTE are real-time and performance properties. However, the definition of power consumption rates is possible as well. Arpinen et al. presented an extension for the MARTE profile (Arpinen, Salminen, Hämäläinen, & Hännikäinen, 2010) that allows defining power state machines for hardware devices that are similar to the energy state charts used in CCM for energy assessment of hardware devices. Nevertheless, they cover the behavior of SW components only by mapping use cases to system states and have no template mechanism for reuse. A further approach using UML and MARTE for the modeling of systems that can be reconfigured w.r.t. NFPs at runtime has been presented by Quadri et al. (Quadri, Gamatié, Boulet, & Dekeyser, 2010). Although they mention power consumption as an NFP, the details of their power consumption model are not further elaborated. Furthermore, their approach focuses on hardware-near embedded systems whereas our approach focuses on component-based software applications deployed on sever landscapes.

## Resource Utilization and Energy Assessment of Software Applications

The research of software's resource utilization or even its energy consumption is a challenging research topic and an important prerequisite for EAT.

A general introduction into energy-aware computing and different aspects that influence the energy consumption of software and hardware applications can be found in (Ellis, 2007). An introduction into performance evaluation and profiling can be found in (Fortier & Michel, 2003).

Lafond et al. investigated how to predict the average energy consumption of Java-based applications. They profiled Java bytecode instructions by executing them on a specific JVM and hardware landscape (Lafond & Lilius, 2006). They were able to measure and predict the energy consumption rate for a large subset of the Java bytecode instruction set and evaluated their results on several benchmarks.

Similar work has been done by Seo et al. (Seo, Malek, & Medvidovic, 2007) (Seo, Malek, & Medvidovic, 2008). Again, the energy consumption of Java bytecode instructions was profiled resulting in a framework for the prediction of Java applications' energy consumption. Besides profiling and prediction, the predicted values were compared with actual consumption rates by measuring the consumption through the application's execution resulting in a variance of less than five percent between predicted and measured energy consumption rates.

Another framework for the resource utilization analysis of Java applications has been built by Navas et al. (Navas, Méndez-Lojo, & Hermenegildo, Customizable resource usage analysis for Java bytecode - Deliverable 2.6 - Preliminary report on advanced resource policies, 2006) (Navas, Mendez-Lojo, & Hermenegildo, Safe upper-bounds inference of energy consumption for Java bytecode applications, 2008). They developed a framework that employs formal methods and control-flow graph analysis to derive formulas for a Java application's resource utilization depending on its input parameters. The user has to specify expected formulas for the NFPs he is interested in (e.g., an expected correlation of a method's input parameters and its energy consumption) and the framework will optimize this formula resulting in a more precise prediction for the NFPs. The framework's energy prediction is based on the results from Lafond et al. for energy prediction (Lafond & Lilius, 2006). The approach of Navas et al. can be compared to our approach for a Java program's resource utilization analysis. However, our approach uses profiling able to derive formulas for resource utilization without hints specified by the user.

A further approach w.r.t. profiling and prediction of applications' energy consumption has been presented by Kansal et al. in 2008 (Kansal & Zhao, 2008). They developed the Windows tool JouleMeter that allows predicting the energy consumption of processes running on a Windows desktop PC. The tool is calibrated using either the battery sensor of the PC running in battery power mode or an external power measuring tool when running on a continuous power supply. Afterwards, JouleMeter is able to estimate the power consumption of processes by profiling their resource utilization (e.g., CPU usage and memory allocation).

## FUTURE WORK

Regarding future work and remaining challenges for EAT we discuss three important parts in the following. These comprise extensions at the user layer of THEATRE, improved simulation and profiling mechanisms and an evaluation of EAT.

Currently, our starting point for energy optimization is a single feature request to an EAT-enabled application. Such requests are combined with quality requirements of a single user in a textual language. However, in realistic distributed applications there is a multitude of users having different quality requirements. Considering each single user for EAT in such a case is not manageable at runtime. Hence, we plan to investigate multi-user models in the future.

A key challenge for energy auto-tuning is to estimate the energy consumption of an application for a given user request. For this purpose we use ESCs in combination with the JRUPI framework. An important drawback of ESCs is that they do not support parallelism. Instead the simulation of multiple ESCs in parallel has to be transformed into sequential execution traces. Although for small amounts of only few ESCs this transformation does not pose a problem, it does not scale. As an alternative, we plan to investigate Petri nets (Petri, 1962), because they provide means for parallel execution and a variety of tools to simulate Petri nets already exists.

An important limitation of JRUPI in its current state of development is that the resource usage model cannot be used to derive fine-grained workloads (e.g., for the ESC simulator or Petri nets). This is because JRUPI considers resource usage in general and not over time. This may lead to an inaccurate workload and in consequence to variances of error in the simulation result. Therefore, we plan to extend JRUPI in a sense that extracted resource usage models are based on time, so that the workload for the simulator can be derived more accurately. That is the offline analysis of JRUPI will be extended to answer the question, which resources have been utilized over which periods of time. Another important drawback of this approach is that resource usage models depend on concrete resources. Imagine two servers and a software component variant that writes a certain amount of data to a HDD. The first server has just a single HDD, whereas the second server has multiple HDDs organized in a RAID system. In order to write the same amount of data by the software component, the two servers will have different resource usage statistics, because different resources are utilized. Hence, for each server a separate resource usage model needs to be determined (as long as they do not consist of completely identical hardware). Consider a software component having N implementations and M servers (with different configurations). To fully cover such a HW/SW system, N * M resource usage models need to be determined. JRUPI uses non-functional benchmarks written by the component developers to utilize the

resources for profiling in a meaningful way. We plan to automate the process of benchmark execution, profiling and resource usage model determination, so all needed models can be computed on the target (i.e., productive) infrastructure once per server before the EAT system is started the first time for productive use.

A prototype of THEATRE (based on OSGi) and a modeling tool suite for CCM and ECL (based on Eclipse) have already been developed. Nevertheless, to evaluate our EAT approach we need to measure the energy consumption of the whole IT infrastructure as well as of single hardware resources. Measured results have to be compared with the predicted energy consumption based on CCM models, ECL contracts and user requests. We started the implementation of an energy-consumption profiling and testing framework for Java applications (Wilke, Götz, Reimann, & Aßmann, 2011) that will be further improved and extended in future work. Another important point is to consider the energy consumption caused by energy auto-tuning itself. This comprises energy effects due to the optimization process itself, energy effects due to system reconfiguration as well as energy effects due to collecting data about the EAT system at runtime. Results of this evaluation should be considered in further refinements of EAT mechanisms.

As a case study we are implementing the stock tracking application described in section "Architecture of Energy Auto-Tuning Systems". Further, we plan to implement a second case study based on a video application. Such energy-aware applications are executed in the THEATRE, whose basic infrastructure (i.e., corresponding managers) is implemented as OSGi components (OSGi Alliance, 2011). So far, resource managers are able to collect resource specific data at server level and to deliver an infrastructure model to the global energy manager. Energy managers are able to monitor deployed software component types and software components which are OSGi bundles containing specific metadata. User managers are not considered so far as we are currently focusing on a single user model. For this purpose the already implemented textual request language is sufficient. The energy auto-tuning mechanisms explained in this chapter are also implemented prototypically. Hence, the current implementation of THEATRE allows to use data from other layers and to execute the EAT control loop except for the acting phase. The latter requires an interpreter for the reconfiguration plan that is part of current work. In fact, it is possible to estimate the best software/hardware mapping of a given application based on simplified assumptions regarding the energy consumption of software components.

## CONCLUSION

This chapter provided an overview of our approach for energy auto-tuning (EAT), which allows for self-optimization w.r.t. the tradeoff between the contradicting goals of reducing energy consumption and improving user utility. It enables the modeling and development of complex hardware/software systems with special focus on NFPs. We elaborated on the architectural elements of EAT systems. These include the CCM, our component model to capture software components and resources of the underlying IT infrastructure as well as their energy-related behavior and the ECL, our contract language to express dependencies between software components and between software components and resources in terms of NFPs. Furthermore, we discussed the mechanisms required by EAT systems. These include

(1) Utility assessment by contract checking: determining system configurations, which are able to fulfill the utility requirements of the user,

(2) Contract negotiation: which aims to derive the optimal of the previously identified system configurations,

(3) Reconfiguration: which derives a plan of required actions to switch from one configuration to another, and

(4) Energy assessment: answering the question how much energy will be consumed by invoking a feature in a certain system configuration.

The EAT approach covers a wide range of research questions from different areas: Profiling energy usage of feature invocations, simulation and self-optimization as well as -adaptation to name but a few. Not all research questions w.r.t. EAT system have been discussed conclusively as we have shown in the previous section. We plan to do further research in these areas in the research projects CoolSoftware, ZESSY/QualiTune and the Collaborative Research Center HAEC, which integrates research groups from electrical engineering and computer science.

## Acknowledgements

This research is part of the projects ZESSY #080951806, CoolSoftware #FKZ13N10782 and the DFG-funded Collaborative Research Center 912 (HAEC). The ZESSY project is funded by the European Social Fund and Federal State of Saxony. The project CoolSoftware is part of the Leading-Edge Cluster „Cool Silicon“, which is sponsored by the Federal Ministry of Education and Research (BMBF) within the scope of its Leading-Edge Cluster Competition.

## REFERENCES

Armengaud, E., Zoier, M., Baumgart, A., Biehl, M., Chen, D., Griessnig, G., et al. (2011). Model-based toolchain for the efficient development of safety-relevant automotive embedded systems. *SAE 2011 World Congress & Exhibition* .

Arpinen, T., Salminen, E., Hämäläinen, T., & Hännikäinen, M. (2010). Extension to MARTE profile for modeling dynamic power management of embedded systems. In *1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED), co-located with DATE 2010, Dresden, Germany, March 12th, 2010.*

Baumgart, A. (2010). A common meta-model for the interoperation of tools with heterogeneous data models. *ECMFA 2010 - 3rd Workshop on Model-Driven Tool & Process Integration* .

Baumgart, A., Reinkemeier, P., Rettberg, A., Stierand, I., Thaden, E., & Weber, R. (2010). A model-based design methodology with contracts to enhance the development process of safety-critical systems. In *Software Technologies for Embedded and Ubiquitous Systems (Vol. 6399 of LNCS)* (pp. 59-70). Springer, Berlin / Heidelberg.

Fei, Y., Zhong, L., & Jha, N. K. (2008). An energy-aware framework for dynamic software management in mobile computing systems. *ACM Transactions on Embedded Computer Systems* (7), pp. 1-31.

Fleurey, F., & Solberg, A. (2009). A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *Model driven engineering and systems (Vol. 5795 of LNCS)* (pp. 606-621). Berlin / Heidelberg: Springer.

Flinn, J., & Satyanarayanan, M. (2004). Managing battery lifetime with energy-aware adaptation. *ACM Transactions on Computer Systems* (22), pp. 137-179.

Fortier, P. J., & Michel, H. E. (2003). *Computer systems performance evaluation and prediction.* Burlington, MA: Digital Press.

Frigo, M., & Johnson, S. G. (1998). FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics Speech and Signal Processing* (pp. 1381-1384). IEEE.

Gartner, Inc. (April 2007). Gartner estimates ICT industry accounts for 2 percent of global CO2 emmisions. *Press Release* .

Geihs, K., Khan, M. U., Reichle, R., Solberg, A., Hallenstein, S., & Merral, S. (2006). Modeling of component-based adaptive distributed applications. *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing* , pp. 718-722.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming* (8), pp. 231-274.

Hewlett-Packard; Intel; Microsoft; Phoenix Technologies; Toshiba. (2010). Advanced configuration and power interface specification, revision 4.0a.

Kansal, A., & Zhao, F. (2008). Fine-grained energy profiling for power-aware application design. In *First Workshop on Hot Topics in Measurement and Modeling of Computer Systems (HotMetrics08) at ACM Sigmetrics.* ACM Press.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., & Loingtier, J.-M. (1997). Aspect-oriented programming. In *ECOOP'97 - Object-Oriented Programming (Vol. 1241 of LNCS)* (pp. 220-242). Berlin / Heidelberg: Springer.

Lachenmann, A., Marrón, P. J., Minder, D., & Rothermel, K. (2007). Meeting lifetime goals with energy levels. *Proceedings of the 5th international conference on Embedded networked sensor systems* (pp. 131-144). ACM Press.

Lafond, S., & Lilius, J. (2006). An energy consumption model for an embedded Java virtual machine. *Architecture of Computing Systems - ARCS 2006 (Vol. 3894 of LNCS)* , pp. 311-325.

Morin, B., Barais, O., Nain, G., & Jézéquel, J.-M. (2009). Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)* (pp. 122-132). Washington, DC, USA: IEEE Computer Society.

Navas, J., Méndez-Lojo, M., & Hermenegildo, M. (2006). *Customizable resource usage analysis for Java bytecode - Deliverable 2.6 - Preliminary report on advanced resource policies.*

Navas, J., Mendez-Lojo, M., & Hermenegildo, M. (2008). Safe upper-bounds inference of energy consumption for Java bytecode applications. *Proceedings of The Sixth NASA Langley Formal Methods Workshop* , pp. 29-32.

Object Management Group (OMG). (2011). *UML profile for MARTE: Modeling and analysis of real-time embedded systems, version 1.1.* Retrieved September 9, 2011, from http://www.omg.org/spec/MARTE/

Oracle. (2010). *BTrace user's guide*. Retrieved July 21, 2011, from http://kenai.com/projects/btrace/pages/UserGuide

OSGi Alliance. (2011). *OSGi service platform release 4*. Retrieved July 29, 2011, from http://www.osgi.org/Release4/

Petri, C. A. (1962). *Kommunikation mit Automaten.* Bonn: Institut für instrumentelle Mathematik der Universität Bonn.

Quadri, I., Gamatié, A., Boulet, P., & Dekeyser, J. (2010). Modeling of configurations for embedded system implementations in MARTE. In *1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED), co-located with DATE 2010, Dresden, Germany, March 12th, 2010.*

Ranganathan, P., & Nygard, K. (2010). An optimal resource assignment problem in smart grid. In *Proceedings of the Second International Conference on Future Computational Technologies and Applications (FUTURE COMPUTING)* (pp. 28-34). Lisbon, Portugal: IARIA.

Schmidt, M., & Lipson, H. (2009). Distilling free-form natural laws from experimental data. *Science* (Vol. 324, no. 5923), pp. 81-85.

Seo, C., Malek, S., & Medvidovic, N. (2007). An energy consumption framework for distributed Java-based systems. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, Atlanta, Georgia, USA* .

Seo, C., Malek, S., & Medvidovic, N. (2008). Estimating the energy consumption in pervasive Java-based systems. In *Proceedings of the 2008 sixth Annual IEEE International Conference on Pervasive Computing and Communications.*

Srikantaiah, S., Kansal, A., & Zhao, F. (2008). Energy aware consolidation for cloud computing. In *Proceedings of the 2008 conference on power aware computing and systems.*

Sun. (2008). *Solaris dynamic tracing guide.* Retrieved May 10, 2011, from http://download.oracle.com/docs/cd/E19253-01/817-6223/

The Climate Group. (2008). SMART 2020: Enabling the low carbon economy in the information age. *Report on behalf of the Global eSustainability Initiative (GeSI)* .

The SPEEDS Consortium. (2009). *D.2.1.5 SPEEDS L-1 Meta-Model.* Retrieved from http://speeds.eu.com/downloads/SPEEDS Meta-Model.pdf

Tsirogiannis, D., Harizopoulos, S., & Shah, M. A. (2010). Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 International Conference on Management of Data (SIGMOD '10)* (pp. 231–242). New York, NY, USA: ACM.

Whaley, R. C., & Dongarra, J. (1997). *Automatically tuned linear algebra software.* Knoxville: University of Tennessee.

## ADDITIONAL READING

Aagedal, J. Ø. (2001). *Quality of service support in development of distributed systems, PhD thesis.* University of Oslo.

Barroso, L. A., & Hölzle, U. (2007). The case for energy-proportional computing. *IEEE Computer , 40* (12), pp. 33-37.

Benini, L., Hodgson, R., & Siegel, P. (1998). System-level power estimation and optimization. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED-98),* (pp. 173–178). New York: ACM Press.

Beugnard, A., Jézéquel, J.-M., & Plouzeau, N. (2010). Contract aware components, 10 years after. *Electronic proceedings in theoretical computer science* , pp. 1-11.

Beugnard, A., Jézéquel, J.-M., Plouzeau, N., & Watkins, D. (1999). Making components contract aware. *IEEE Computer* (32), pp. 38-45.

Borkar, S., & Chien, A. A. (2011). The future of microprocessors. *Communications of the ACM , 54*, pp. 67-77.

Chung, L., & do Prado Leite, J. C. (2009). On non-functional requirements in software engineering. In *Conceptual Modeling: Foundations and Applications (Vol. 5600 of LNCS)* (pp. 363-379). Berlin / Heidelberg: Springer.

Chung, L., Nixon, B. A., Yu, E., & Mylopoulos, J. (2000). *Non-functional requirements in software engineering.* Norwell, MA, USA: Kluwer Academic Publishers.

Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2007). *Algorithms.* Columbus, OH, USA: McGraw-Hill Higher Education.

Dobson, S., Denazis, S., Fernández, A., Gärti, D., Gelenbe, E., Massacci, F., et al. (2006, 1). A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems* , pp. 223-259.

Ellis, C. (2007). *Controlling energy demands in mobile computing systems (Synthesis lectures on mobile and pervasive computing).* Morgan and Claypool Publishers.

Göbel, S., Pohl, C., Röttger, S., & Zschaler, S. (2004). The COMQUAD component model: enabling dynamic selection of implementations by weaving non-functional aspects. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, Lancaster, UK, March 22 - 24, 2004* , pp. 74-82.

Götz, S., Wilke, C., Cech, S., & Assmann, U. (2011). To appear: Runtime variability management for energy-efficient software by contract negotiation. In *Proceedings of 6th International Workshop Models@run.time.*

Götz, S., Wilke, C., Schmidt, M., Cech, S., & Assmann, U. (2010). Towards energy auto tuning. *Proceedings of First Annual International Conference on Green Information Technology (GREEN IT)* (pp. 122-129). Singapore: GSTF.

Götz, S., Wilke, C., Schmidt, M., Cech, S., Waltsgott, J., & Fritzsche, R. (2010). *THEATRE Resource Manager Interface Specification v. 1.0, Technical Report TUD-FI10-08.* Technische Universität Dresden.

Jacob, B., Larson, P., Leitao, B. H., & da Silva, S. A. (2010). *SystemTap: Instrumenting the Linux kernel for analyzing performance and functional problems.* IBM Redbooks.

Meyer, B. (1992). Applying "design by contract". *Computer , 25* (10), pp. 40-51.

Meyer, B. (1997). *Object-oriented software construction* (2 ed.). Prentice Hall.

Nemhauser, G. L., & Wolsey, L. A. (1988). *Integer and combinatorial optimization.* New York, NY, USA: Wiley-Interscience.

Püschel, G. (2011). Energieeffizienz in Workflowsystemen, Diploma Thesis. Technische Universität Dresden.

Rivoire, S. M. (2008). *Models and metrics for energy-efficient computer systems, PhD thesis.* Stanford University.

Röttger, S., & Zschaler, S. (2003). Enhancements to CQML. *Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering* , pp. 43-56.

Süttner, P. (2011). Abstrakte Verhaltensbeschreibung von CCM Softwarekomponenten, Diploma Thesis. Technische Universität Dresden.

Szyperski, C., Gruntz, D., & Murer, S. (2002). *Component software - Beyond object-oriented programming.* Addison-Wesley / ACM Press.

Wilke, C., Götz, S., Reimann, J., & Aßmann, U. (2011). Vision paper: Towards model-based energy testing. In *Proceedings of the ACM/IEEE 14th International Conference on Model Driven Languages and Systems (MODELS2011), Wellington, New Zealand, October 16-21, 2011 (Vol. 6981 of LNCS)* (pp. 480-489). Berlin / Heidelberg: Springer.

## KEY TERMS & DEFINITIONS

| | |
|---|---|
| Non-functional Property: | A characteristic, which reflects *how* a component operates. For example, the response time, expressing how fast an operation of a component is. |
| Quality Contract: | Defines dependencies between software and hardware components in terms of provided and required non-functional properties. |
| Contract Checking: | The process of determining all system configurations, which fulfill the constraints defined in quality contracts. |
| Contract Negotiation: | The process of computing the optimal system configuration, which serves the user's request, ensures the user's minimum requirements and requires the least energy. |
| User utility: | A function mapping non-functional properties – which are perceptible by the user (e.g., framerate) – to a finite set of levels of satisfaction (e.g., video playback is perceived by the user as fluent, weak hesitant or strong hesitant). |
| System configuration: | A selection of (software) component implementations and their mapping onto system resources. |
| Energy assessment: | The process of determining how much energy is required by a certain feature of the system in a specified system configuration. |
| Self-reconfiguration: | The ability of a (component-based) system, to adjust itself at runtime by changing its configuration at runtime. |